Custom Processor Design for Efficient, yet Flexible Lucas-Kanade Optical Flow

Sander Smets*, Toon Goedemé† and Marian Verhelst*
*ESAT - MICAS
KU Leuven, Leuven, Belgium
†EAVISE - PSI
KU Leuven, Campus De Nayer, Sint-Katelijne-Waver, Belgium
Email: { sander.smets, toon.goedeme, marian.verhelst}@esat.kuleuven.be

***Abstract*—State-of-the-art solutions to optical flow fail to jointly offer high density flow estimation, low power consumption and real time operation, rendering them unsuitable for embedded applications. Joint hardware-software scalability at run-time is crucial to achieve these conflicting requirements in one device. This paper therefore presents a scalable Lucas-Kanade optical flow algorithm, together with a flexible power-optimized processor architecture. The C-programmable processor exploits algorithmic scalability through innovations in its memory structure, memory interface, and datapath optimized for efficient convolutions. Jointly, the scalable flow algorithm and optimized computer vision hardware platform enable applications to on-the-fly trade-off throughput and power consumption in function of flow density and accuracy. The processor chip is synthesized in 40nm CMOS technology and verified on FPGA. The architecture is capable of scaling the frame rate at run-time and processes 16fps of dense optical flow at 640×480 resolution with 15.06° average angular error, while only consuming 24mW.**

Keywords–Machine vision, programmable circuits, parallel architectures, real time systems, low-power electronics.

I. Introduction

Autonomous visual navigation is an up and rising topic in industry. To increase safety and efficiency, cars, robots and drones are navigating more and more autonomously. This requires vehicles to be aware of their environment and process their on-board sensors in real-time. This should be done at low power consumption, especially for battery-powered, small form-factor devices, such as e.g. a nano drone.

Optical flow estimation has a key role in visual navigation tasks like collision detection [1] and ego-motion estimation [2]. Tracing pixels between frames in a video sequence enables motion-based image processing, taking the time aspect of a video sequence into account. Certain motion-based tasks, such as collision detection, require a dense optical flow field. In dense optical flow estimation, the flow field is typically estimated for more than 40% of all pixels. For other tasks, such as ego-motion estimation, tracking a small set of key points suffices for reliable operation. While the number of key points may be several thousands, the flow density remains well below 1% of all pixels. Since embedded devices may require both dense and sparse flow fields, the contrast in field density calls for scalable solutions which are flexible regarding the optical flow parameters, yet remain extremely efficient. For visual navigation, for example, sparse optical flow can be performed until a dangerous or interesting pattern is detected. Upon such event, dense optical flow fields can be derived in certain image sections, for very precise navigation (e.g. in case of collision danger).

Dense and sparse optical flow have been studied for a long time and have been implemented in different ways. Many state-of-the-art algorithms are executed on CPUs, because of their ease of programming. However, such general purpose platforms have a negative impact on the optical flow throughput and execution times of several minutes per frame are no exception [3].

GPUs counter this problem by using widely-parallel data paths (e.g. [4]): the same operations are executed on many pixels simultaneously. As a result, GPUs allow to compute dense optical flow in real-time. However, the price to pay is a significant increase in power consumption. GPUs consume tens of watts, rendering them unfit for small, battery-driven embedded devices.

Alternatively, dedicated hardware on Field-Programmable Gate Arrays (FPGAs) has been designed (e.g. [5]–[7]). These implementations allow high optical flow throughputs at relatively low power consumption. Yet, the drawback of these designs is their lack of on-the-fly adaptivity. They achieve good performance and efficiency by implementing dedicated data paths, highly optimized for a particular optical flow algorithm and settings. As such, they are unfit to efficiently support various frame sizes, flow densities or other algorithm adaptations.

To enable adaptive optical flow on small form-factor devices, we propose an Application-Specific Instruction-set Processor (ASIP) architecture. This is a flexible, yet efficient C-programmable processor, optimized for optical flow computations. In this paper, we will describe the processor's architecture and show how it is capable of executing various flavors of optical flow at low power and sufficiently high throughput.

The rest of this paper is structured as follows: in Section II the framework of scalable optical flow is described. Section III presents the designed architecture, which is evaluated and compared to the current state-of-the-art in Section IV. Section V concludes this paper and indicates future work on this topic.

II. A scalable dense/sparse optical flow algorithm

The computation of dense and sparse flow fields is commonly seen as two separate problem statements. Consequently, solutions are dedicated to a single scenario and not very suited towards scalability. In general, sparse optical flow estimations rely mainly on key point description and matching (e.g. [8]).

The work of S. Smets was supported by a Doctoral Fellowship of the Research Foundation Flanders (FWO).

In contrast, dense optical flow algorithms generally consist of global iterations, refining the flow for every pixel over the entire image (e.g. [9]). Recent advances in dense flow like EpicFlow [10] combine both approaches to achieve even better accuracies, but compromise on computation time in the process.

We provide an alternative by extending the algorithm of Lucas and Kanade [11] towards scalability in terms of denseness and accuracy. Although Lucas-Kanade has been outperformed by more recent techniques in terms of accuracy, its underlying principles estimate flow through local computations, making it a good candidate for adaptive flow densities. Additionally, we will show that the pyramidal approach introduced by Bouguet [12] offers a way to trade accuracy for throughput.

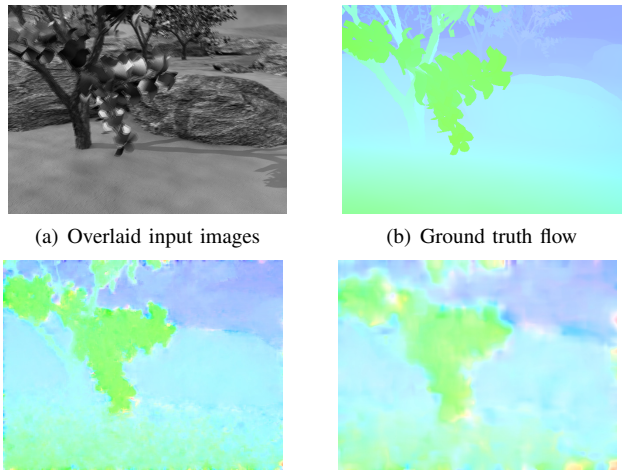
Fig. 1 contains the pseudo-code for our scalable version of pyramidal Lucas-Kanade flow estimation. Note that flow density (d) and the scales processed in the pyramidal approach (s_{start} to s_{stop}) are input values and indicate the two introduced scalability parameters. These two parameters impact the workload significantly.

Firstly, the flow density d determines the number of tracked points. Tracking more points amounts to a larger workload (Fig. 1: more iterations at line 15), hence reducing the frame rate. However, as the flow field becomes more dense, key points approach each other, giving rise to common operations. At a specific threshold, the overhead introduced by these common operations justifies the computation of a fully dense flow field, where intermediate results can be re-used (Fig. 1: test at line 2). Furthermore, our dense flow model adopts the acceleration by Integral Images as presented in [13] and implements a pixel-wise image warping technique, rather than extracting interpolated neighborhoods for every individual pixel. Note that Plyer et al. [14] indicate that the adoption of such warping scheme makes the algorithm unstable for multiple iterations. Since we only consider one iteration per scale for the rest of this paper, this does not affect our implementation. However, if multiple iterations are desired, our programmable platform allows to perform the required adaptations at a software level.

Input: Input images I_1, I_2, flow density d, scale range between s_{start} and s_{stop}
Output: Optical flow estimation u, v

1: Generate Gaussian pyramids for I_1 and I_2
2: **if** $d >$ threshold **then**
3: {Dense flow}
4: initialize flow at 0
5: **for** $i = s_{start}$ to s_{stop} **do**
6: warp $I_{i,2}$ according to current flow estimation
7: convolve for image derivatives $d_{i,x}$, $d_{i,y}$ and $d_{i,t}$
8: multiply for cross-products $I_{i,xx}$, $I_{i,xy}$, $I_{i,yy}$, $I_{i,xt}$ and $I_{i,yt}$
9: sum $I_{i,xx}$, $I_{i,xy}$, $I_{i,yy}$, $I_{i,xt}$ and $I_{i,yt}$ over 7×7 neighborhood
10: calculate flow u_i and v_i
11: **end for**
12: **else**
13: {Sparse flow}
14: convolve for image derivatives $d_{i,x}$, $d_{i,y}$ at every pyramid scale i
15: **for** $k = 1$ to d **do**
16: initialize flow for keypoint k at 0
17: **for** $i = s_{start}$ to s_{stop} **do**
18: extract neighborhoods
19: multiply for cross-products $I_{i,xx}$, $I_{i,xy}$, $I_{i,yy}$, $I_{i,xt}$ and $I_{i,yt}$
20: sum $I_{i,xx}$, $I_{i,xy}$, $I_{i,yy}$, $I_{i,xt}$ and $I_{i,yt}$ over 7×7 neighborhood
21: calculate flow u_i and v_i for keypoint k
22: **end for**
23: **end for**
24: **end if**

Figure 1. Pseudo-code for scalable Lucas-Kanade optical flow.



(a) Overlaid input images (b) Ground truth flow

(c) Calculated flow (4 out of 4 scales) (d) Calculated flow (2 out of 4 scales)

Figure 2. Dense optical flow results for the Grove2 image from the Middlebury benchmark. Fig. 2(c) and 2(d) show the output from the Lucas-Kanade dense flow estimation for two operation modes with different accuracy settings.

Besides the density scalability, a second scaling factor is determined by the relationship between the number of image scales and the flow accuracy: a higher number of scales typically results in high accuracy levels, but corresponds to more computations (Fig. 1: more iterations at lines 5 and 17). Fig. 2 shows the inputs and outputs for the Grove2 images from the Middlebury dataset [15], subject to our software model at different settings. Fig. 2(d) shows the output when only two pyramid levels are taken into account ($s_{start} = 4$, $s_{stop} = 3$), whereas for Fig. 2(c) four scales were processed ($s_{start} = 4$, $s_{stop} = 1$). Notice that, even though the two-scale flow image is not as accurate as its four-scale counterpart, it gives a good indication of the flow field and may be sufficient for embedded applications. The trade-off between accuracy and frame rate offers a second degree of freedom for applications to change their operating point.

For a more detailed overview of the algorithm, we refer to our open-source matlab implementation, available on our website [16].

Note that a programmable platform allows numerous techniques to extend this software model besides the parameter variations discussed above. The introduction of a Region of

Interest (RoI), for example, offers a frame rate increase at the cost of only processing a part of the input image. These techniques will not be considered for the rest of this paper, but since the designed processor is fully C-programmable, their implementation can be realized at a software level.

III. A FLEXIBLE PROCESSOR FOR SCALABLE OPTICAL FLOW

In this work, we present a Reduced Instruction-Set Computer (RISC), enhanced with custom instructions to accelerate optical flow. The high-level architecture is shown in Fig. 3. We present four innovations which increase throughput and energy-efficiency, while maintaining flexibility: widely-parallel data paths (Section III-A), an image processing adapted memory organization (Section III-B), a warping module (Section III-C) and custom convolution hardware (Section III-D).

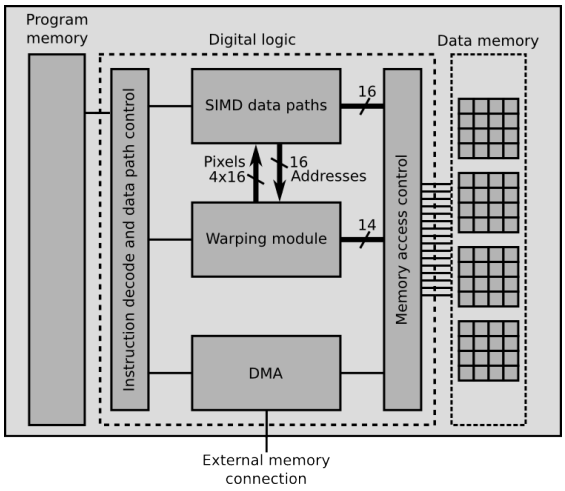


Figure 3. The overall platform architecture.

A. Parallelism and word length optimized data path

The targeted class of algorithms execute the same operations on each pixel. This allows to take advantage of Single-Instruction Multiple-Data (SIMD) instructions. Every SIMD operation processes multiple pixels in parallel, while the instruction only has to be fetched and decoded once. Ideally, this increases the throughput by a factor n, with n being the number of parallel data paths. Simultaneously, the energy consumption spent in fetching and decoding can be decreased by a factor n.

Notice that only operations with inherent pixel parallelism can profit from the SIMD parallel data paths. For control operations, such as loop management or flow control, or irregular data accesses this is typically not the case. The optimizations in sections III-B to III-D seek to reduce this overhead and maximize the utilization rate of the parallel functional units.

Our architecture contains 16 data paths, each with a 16-bit fixed point data path unit. The data path word length was chosen based on a word length study. Fig. 4 shows the influence of the number of bits on the Average Angular Error [17] of the dense optical flow estimation. From this study, it

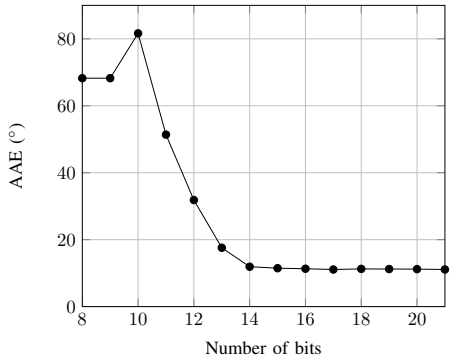


Figure 4. Average Angular Error over the entire Middlebury dataset [15] as a function of the word lengths for scaled images. The last value is the error achieved for double floating point computations.

can be seen that as soon as the image pyramid is represented with less than 14 bits, the output starts to degrade. Increasing the number of bits, however, does not have a significant impact on the output values. In this case, the quantization error introduced by short word lengths is negligible compared to the accuracy of the Lucas-Kanade algorithm. The increased hardware cost and power consumption of wider word lengths on the one hand, and the common practice of having a power of two for word lengths on the other, motivate the choice for a 16-bit architecture.

Note that further processing steps of the algorithm require longer fixed point word lengths (e.g. the representation of the integral images). An overview of all word lengths is shown in Table I. To cope with these higher word lengths while using the 16-bit adders and multipliers implemented in the processor, support was added to the processor's instruction set to support multi-cycle, multi-word operations.

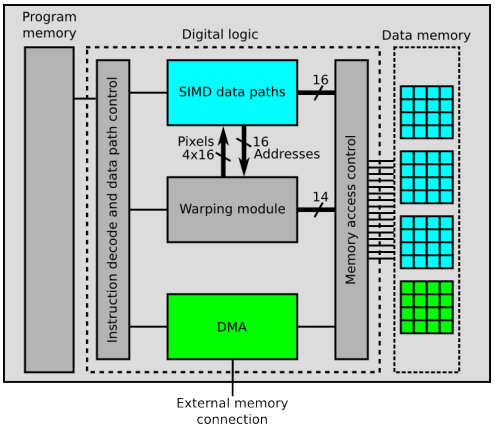
TABLE I. FIXED POINT WORD LENGTHS THROUGHOUT THE OPTICAL FLOW COMPUTATIONS. b_i AND b_f REPRESENT THE NUMBER OF BITS USED FOR THE INTEGER AND THE FRACTIONAL PART RESPECTIVELY.

Data	*Number of bits (b_i,b_f)*
Images	(8,8)
Derivatives	(8,8)
Cross-products	(16,16)
Integral images	(22,10)
System coefficients	(44,20)
Output flow	(8,8)

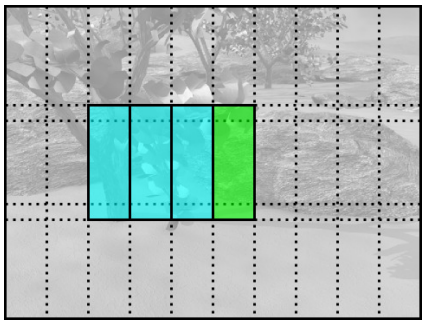
B. Image processing adapted memory structure

To extend the parallelism from the data paths towards the memory storage operations, a single load/store instruction should correspond to 16 parallel load/store operations. Therefore our architecture contains a 128kB data memory, split over 64 memory banks. Every bank has one read/write port, allowing for one memory transfer per clock cycle.

Since 128kB is not sufficient to contain the input images and all intermediate results, the images are processed in slices of 16 pixels wide and operations are parallelized across rows of 16 pixels. However, calculating filter outputs requires patches around the corresponding input pixels. For output values on the edge of an image slice, these areas contain pixels from



(a) Memory access of platform blocks.



(b) Image slices with overlap in vertical direction.

Figure 5. Data memory division in 4 blocks. The SIMD data paths have access to 3 blocks of 16 memory banks, while the DMA manages data traffic between the 4th block and the external DRAM.

neighboring slices. This requires the processor data memory to hold three image slices simultaneously accessible: a left slice, a center slice and a right slice (for filter inputs corresponding to output pixels on the left side, the middle and the right side of the slice respectively). Therefore, the data memory consists of four blocks of 16 memory banks, for four image slices. Of these blocks, there are always three accessible in parallel by the processor while the fourth is used by a Direct Memory Access (DMA) module for moving data between the processor and a large external memory. As these roles constantly rotate across the 4 memory blocks, it enables to in parallel load/store data to/from external memory and process data in the SIMD data paths, avoiding processor stalls. This is illustrated in Fig. 5.

The challenge of needing neighboring slices also exists in the vertical direction. Since slices have a limited height, outputs on the top and bottom edges cannot be computed. Therefore, generating image slices that are fit for 2D filtering requires the introduction of overlap between slices, as shown in Fig. 5(b). The amount of overlapping pixels is determined by the vertical filter sizes and should be kept as small as possible to minimize superfluous loading of the same pixels. Note that the number of slices should be kept small as well, for the same reason.

Image processing algorithms regularly require different patterns of input data: horizontal and vertical filters operate on rows and columns respectively, while bilinear interpolations require neighborhoods of 2×2 pixels. If the different pixels are

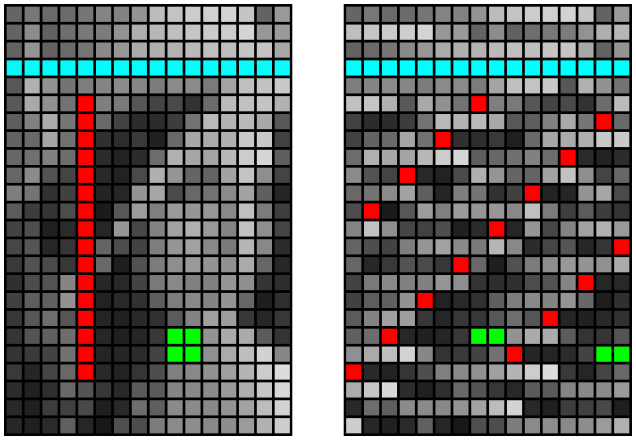


Figure 6. Data re-ordering scheme applied to a single memory block. Left: highly subsampled Lenna image, with image ordering as perceived by the processor. Right: physically re-ordered data where every next row is shifted by 7 positions. Every column represents one memory bank. Notice that rows (cyan), columns (red) and 2x2 environments (green) are spread across different memory banks, independent of their location.

distributed over the 16 memory banks in a linear way, there is no guarantee that these different structures are accessible in parallel. To tackle this problem, we propose an image re-ordering scheme which splits data across different memory banks for every pattern discussed above. This is achieved by determining the physical location where data is stored, by (1):

$$\texttt{bank} = (\texttt{x}_{3..0} + 7 \times \texttt{y}_{3..0}) \pmod{16}. \quad (1)$$

where $\texttt{x}_{i..j}$ and $\texttt{y}_{i..j}$ represent bits j up to i from the pixel x or y coordinate respectively. Fig. 6 shows how a highly subsampled image of Lenna is transformed when this scheme is applied. From Fig. 6, it is clear that the rows, columns and 2×2 environments from the processor's point of view are spread across the memory banks. This always allows parallel extraction of these structures, and accelerates both horizontal and vertical filters, as well as bilinear interpolations.

C. Specialized warping module

The pyramidal Lucas-Kanade approach relies on low-resolution flow estimations to initialize flow at higher-resolution scales. To take this initialization into account, one of the input images is warped according to the current flow estimation. This warping step moves every pixel back to its estimated original location, reducing the size of the remaining flow between the other input image and the warped image.

Since neighboring pixels might not have the same flow values, moving pixels back to their original locations requires highly irregular data accesses. Therefore, the warping step in Lucas-Kanade dense optical flow is extremely memory intensive and can not naturally benefit from the parallelization discussed above. This prevents efficient parallel data fetches and disrupts parallel computations. To still benefit from parallelism during warping, we propose the addition of a warping module in between the banked memory and the processor's register file.

Fig. 7 shows the operating principle of our specialized warping module. It takes 16 addresses from the processor's

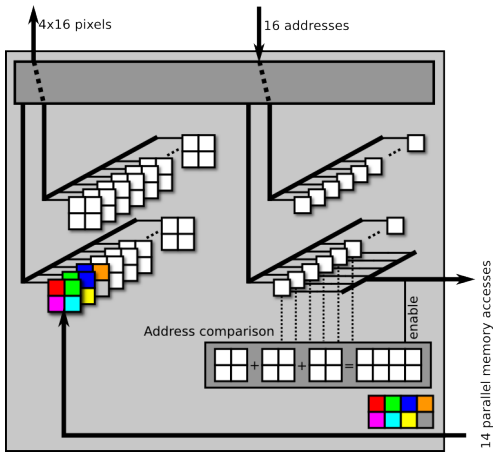


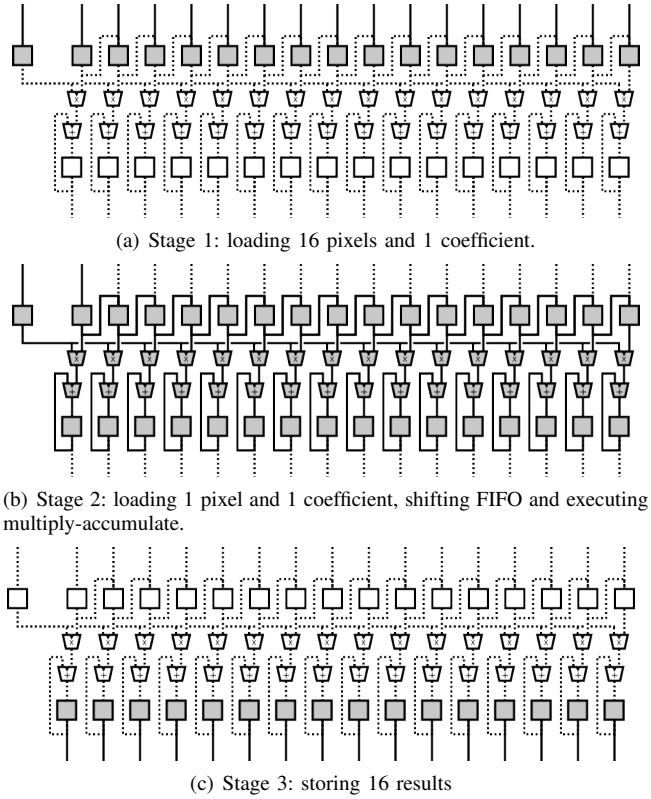
Figure 7. Block diagram for the warping module, in the case where only the first three neighborhoods are adjacent. The module loads a 2×4 neighborhood. The bottom 16×2×2 register structure is used by the warping module, while the upper registers are accessible by the SIMD data paths.

SIMD data paths, fetches the 16 corresponding 2×2 neighborhoods autonomously and stores them in a register file. Note that 2×2 neighborhoods are used for bilinear interpolations to deal with fractional flow values. While the warping module fetches the data, the processor can continue computations on the data previously fetched and issue a new request for the warping module, minimizing processor stalls. As a result, fetching the 4×16 inputs for the bilinear interpolation ideally only takes 1 effective clock cycle. This operational scheme can be interpreted as a Direct Memory Access (DMA) which sits between the on-chip memory and the processor's register file, in analogy with a classical DMA between the processor's external and internal memory.

To minimize stall time in both the SIMD data paths as well as the warping module, the latter consists of two 16×2×2 register structures. The data paths and warping module each have access to one of these structures, and data is communicated between the two in a ping-pong fashion. In Fig. 7, the upper register structure is connected to the SIMD data paths, while the lower part is used for fetching new pixels.

To enhance the throughput of the warping module and balance its computation time with the SIMD data paths, pixels are fetched in parallel. Given the memory structure discussed in Section III-B, the warping module can load at least 4 pixels (one 2×2 block) per cycle. Additionally, the module autonomously checks whether the request contains adjacent pixels in non-conflicting memory banks. Adjacent pixels have the advantage that their 2×2 environments overlap, allowing the warping module to load 3×2 pixels simultaneously. The warping module can exploit this to load up to 7×2 pixels in a single cycle, as this is the largest neighborhood for which memory banks don't overlap. This reduces the number of clock cycles spent on memory fetches, as well as the number of fetches itself, increasing the processor throughput and decreasing the power consumption. Note that neighboring pixels are not guaranteed, yet likely to have similar flow vectors. Consequently, our warping module often loads larger neighborhoods.

It should be noted that the warping module has access



(a) Stage 1: loading 16 pixels and 1 coefficient.

(b) Stage 2: loading 1 pixel and 1 coefficient, shifting FIFO and executing multiply-accumulate.

(c) Stage 3: storing 16 results

Figure 8. The convolution engine in its 3 computation stages.

to the exact same memory banks as the processor itself. If the processor and the warping module would compete over memory access, priority will be given to the processor. However, the shared memory between the processor and the warping module also limits warping distance: as the warping module only has access to three image slices, the module is not guaranteed to be able to warp pixels over more than 16 pixels.

D. Convolution Engine

The fourth adaptation of the processor is the inclusion of a convolution engine in the processor's data path. Convolutions are a computation-heavy component in the computation of Lucas-Kanade optical flow. Therefore the parallel data paths accelerate the computation of 1-dimensional convolutions through custom instructions.

While their regular structure allows convolutions to strongly benefit from parallel computation and parallel memory fetches in a SIMD architecture, they still suffer from inefficiencies. The most important inefficiency is due to neighboring data paths processing the same input data in subsequent clock cycles.

To enable input data sharing, we propose the inclusion of a First In First Out (FIFO)-like register in the data path. For the computation of 16 convolution outputs, the register is filled with the first 16 input pixels in a single clock cycle. Note that the memory structure presented in Section III-B allows for parallel extraction of both rows and columns, enabling both horizontal and vertical filters. After loading the initial pixel values, the processor only loads one new input pixel for every

filter coefficient, while shifting the older pixels (similar to a traditional FIFO).

To further accelerate convolutions, the processor data path is extended with logic to, in parallel: (i) load a new pixel and shift it in the FIFO; (ii) load the next filter coefficient; (iii) perform SIMD multiply-accumulate operations. Note that the convolution engine simultaneously loads a pixel and a filter coefficient. To guarantee that these two operations never access the same memory bank, we add a small (2 kB) coefficient memory with a separate read/write port.

The developed convolution engine executes convolutions in three phases (Fig. 8): (a) filling the FIFO with 16 initial values; (b) performing the convolution while in parallel loading the next coefficient and pixel; and (c) writing back the results. In comparison with a traditional SIMD processor, our design speeds up the calculation of convolutions by a factor 3, while also reducing power consumption in memory operations.

IV. Performance Evaluation

The presented processor was evaluated for chip implementation and functionally verified on FPGA. Section IV-A describes the methodology by which it is evaluated. Section IV-B explains how the ASIP can be programmed and Section IV-C quantifies the enabled trade-off between flow density, accuracy and throughput. Finally, Section IV-D compares the presented approach to state-of-the-art optical flow implementations in terms of throughput and power consumption.

A. Evaluation methodology and platform

a) Chip: To evaluate the performance of the ASIP, the architecture was developed with the Synopsys ASIP Designer and synthesized in 40nm CMOS TSMC technology with the Synopsys tool chain, up until the EDI environment. The placed and routed floorplan of the processor is shown in Fig. 9. The chip measures 1.55 mm $\times$ 1.1 mm and contains 128 kB program and data memories generated by a memory compiler. The maximum clock frequency is 207 MHz.

Table II shows the contributions of all introduced innovations in terms of gate count and power consumption. The power consumption was estimated on an extracted layout after placement and routing, and on a representative part of the dense optical flow algorithm. From the table can be seen that both area and power consumption are largely dominated by the on-chip memories. However, memory reduction is not advised as it would result in smaller image slices (smaller in the vertical direction) and hence more superfluous loading operations.

Within the processing core, the SIMD data paths take up the majority of area and power. When considering only the active power consumption in the core, this effect is even more outspoken: 57.22% of the active power is dissipated in the 16-way SIMD data paths. This indicates that most of the power is spent on key operations in the optical flow algorithm.

TABLE II. Custom chip resource and power consumption for the processor contributions.

Category	*Gates*		*Power consumption*	
Processor control	39451	(1.82%)	0.96 mW	(4.26%)
SIMD data paths	159049	(7.33%)	4.43 mW	(19.74%)
Data and program memory	1865081	(85.94%)	14.84 mW	(66.07%)
Memory structure overhead	33467	(1.54%)	1.52 mW	(6.75%)
Warping module overhead	46665	(2.15%)	0.52 mW	(2.29%)
Convolution engine	26381	(1.22%)	0.20 mW	(0.89%)
Others	10387	(0.48%)	1.53 mW	(6.39%)
Total	2180481		24.0 mW	

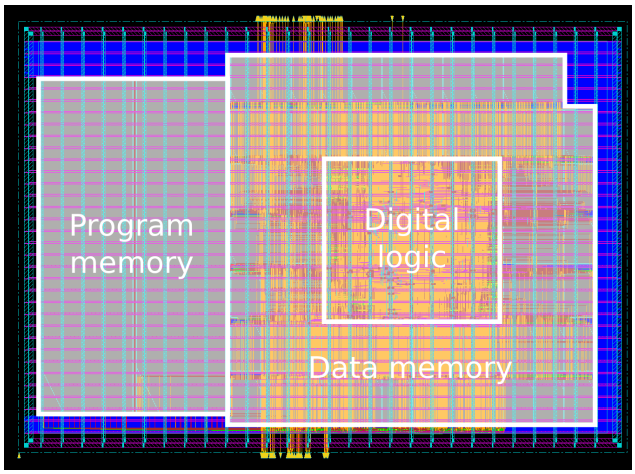


Figure 9. Floor plan of the designed chip, with 128 kB program memory and 128 kB data memory.

b) FPGA: The proposed architecture was verified through synthesis by Xilinx Vivado and has been mapped successfully on the Nexys 4 DDR board. This prototyping board contains an Artix XC7A100T-CSG324 Xilinx FPGA as processing element and an external 128 MiB DDR2 memory. The total used resources are given in Table III.

The maximum clock frequency is 20 MHz, which corresponds to 0.48 frames per second for a fully accurate, fully dense, 640$\times$480 flow field. This clock frequency is an order of magnitude slower than the 40 nm CMOS implementation. Since the memory access control layer connects all memory banks to the SIMD data paths, the critical path is mostly determined by the locations of the Block RAMs on the FPGA. To mitigate this effect, the program memory for the FPGA implementation has been reduced to 32 kB. Though this restricts the processor's flexibility, the program memory is still over-dimensioned by almost a factor 2, leaving room for complex software.

The processor is connected to the development board's external memory through an AXI bus interface and the Vivado Memory Interface Generator. The input data is loaded in the external memory by a JTAG connection to a computer. Through this connection, the calculated flow was extracted from the development board and evaluated on the computer.

B. ASIP programmability

The architecture was programmed in C, using the Synopsys Chess compiler. The Chess compiler is part of the ASIP Designer and allows to define new C functions and maps

TABLE III. FPGA resource utilization for the architecture.

Category	*Utilization*	*Utilization rate*
FF	12599/126800	9.94%
LUT	41853/63400	66.01%
BRAM	43/135	31.85%
DSP48	50/240	20.83%

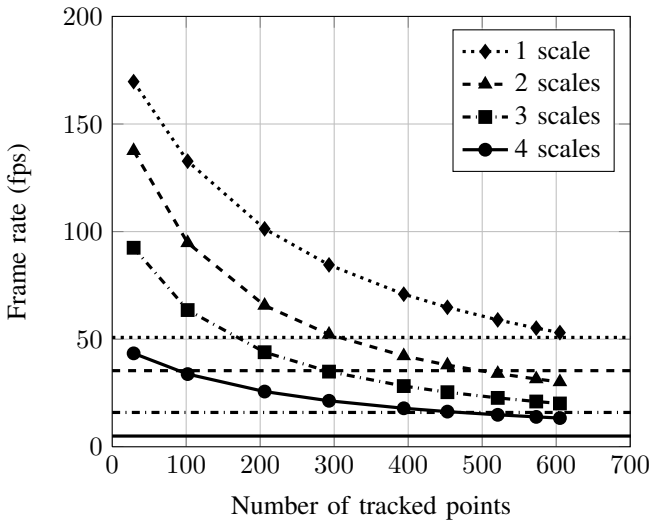


FIGURE 10. FRAME RATE (IN FPS) AS A FUNCTION OF THE NUMBER OF TRACKED FEATURE POINTS. KEY POINTS ARE ASSUMED TO BE SELECTED ON BEFOREHAND. THE 4 CURVES CORRESPOND TO THE 4 ACCURACY SETTINGS USED IN TABLE IV. THE HORIZONTAL LINES CORRESPOND TO FRAME RATES FOR DENSE OPTICAL FLOW. FRAME SIZE IS ASSUMED TO BE 640×480.

these functions to processor instructions. Consequently, vector operations and customized functions like convolutions are reduced to C function calls. Memory management is manually optimized to force image slices into specific memory locations, to ensure the correct positioning of pixels within the memory blocks (see Figure 5).

C. Dynamic throughput-accuracy-density trade-off

The processor's programmability counters the incapability of fixed implementations to support various frame sizes, flow densities or other algorithm adaptations. To evaluate the flexibility of our design and to demonstrate the applicability of algorithmic changes, the optical flow algorithm was evaluated on the Middlebury benchmark [15] for different algorithmic settings. More specifically, the relation between frame rate, accuracy and flow density was mapped.

The number of tracked points, as well as the number of image scales in the pyramidal Lucas-Kanade algorithm has a significant impact on accuracy and throughput. This trade-off between throughput and accuracy is evaluated on the designed processor. In what follows, the input image is considered to be downscaled three times. The computation of optical flow starts at the most downscaled image and scales up the output with every iteration.

TABLE IV. THROUGHPUT AND AVERAGE ANGULAR ERROR FOR DIFFERENT DENSE OPTICAL FLOW VARIANTS. THROUGHPUTS FOR 640×480 IMAGES. COLUMNS CONTAIN THE AAE RESULTS FOR ALL IMAGES IN THE MIDDLEBURY DATASET [15], USING A SPECIFIC ACCURACY SETTING.

	4 scales	*3 scales*	*2 scales*	*1 scale*
Throughput	5.00 fps	15.96 fps	35.38 fps	50.84 fps
AAE				
Dimetrodon	10.15°	15.51°	20.51°	29.37°
Grove2	5.50°	7.88°	11.51°	21.16°
Grove3	10.08°	12.35°	16.05°	23.08°
Hydrangea	8.28°	12.94°	19.32°	26.36°
RubberWhale	18.50°	25.99°	33.28°	40.03°
Urban2	10.16°	12.35°	15.44°	20.96°
Urban3	11.70°	13.93°	17.90°	25.12°
Venus	16.21°	19.53°	24.16°	30.33°
Average	11.32°	15.06°	19.77°	27.05°

The relationship between flow density and throughput originates from the number of points that is tracked. As the computational load rises with the number of key points, flow density can smoothly be traded for throughput. Fig. 10 shows the maximum frame rate as a function of the number of tracked feature points for different accuracy levels. Notice that from a specific number of key points on, it becomes more efficient to perform a fully dense flow than to track individual pixels. This is explained by the re-use of intermediate results in dense optical flow, which is infeasible for sparse feature tracking. Therefore, dense flow should be performed as soon as the desired flow density is larger than a couple hundred key points (the exact number depends on the required accuracy).

Table IV lists the throughput and AAE of the computed flow for all images in the Middlebury benchmark under 100% dense flow computation. As expected, there is a trade-off between throughput and accuracy. As such, our flexible compute platform allows applications to determine an operation point on-the-fly, based on the current requirements (in terms of throughput and/or accuracy).

D. Comparison to the State-of-the-Art

Table V compares the presented processor with State-of-the-Art optical flow implementations. It should be pointed out that for all implementations, the reported processing performances represent the performance of the processing core

TABLE V. COMPARISON OF THE PROPOSED ARCHITECTURE WITH STATE-OF-THE-ART IMPLEMENTATIONS OF OPTICAL FLOW.

Dense flow estimation	*Platform*	*Algorithm*	*Resolution*	*Frame rate*	*Throughput*	*Power*	*Energy / tracked pixel*
This work	**40nm CMOS**	**Pyramidal LK**	**640 × 480**	**5 - 51 fps**	**1.5 - 15.7 MP/s**	**24.0 mW**	**15.6 - 1.54 nJ/pixel**
Murachi et al. [18]	90nm CMOS	Pyramidal LK	640 × 480	30 fps	9.2 MP/s	600 mW	65.2 nJ/pixel
Barranco et al. [**?**]	Pyramidal LK	640 × 480	32 fps	9.8 MP/s	-	-	
Komorkiewicz et al. [19]	FPGA	Horn-Schunck	1920 × 1080	60 fps	175 MP/s	12 W	70.25 nJ/pixel
Plyer et al. [14]	GPU	eFOLKI	640 × 480	112 fps	34.4 MP/s	52 W (*)	1511 nJ/pixel (*)

Sparse flow estimation	*Platform*	*Algorithm*	*Resolution*	*Tracked pixels*	*Frame rate*	*Power*	*Energy / tracked pixel*
This work	**40nm CMOS**	**Pyramidal LK**	**640 × 480**	**29**	**43 - 170 fps**	**24.0 mW**	**19.25 - 4.87 μJ/pixel**
This work	**40nm CMOS**	**Pyramidal LK**	**640 × 480**	**605**	**13 - 53 fps**	**24.0 mW**	**3.05 - 0.75 μJ/pixel**
Sasagawa et al. [20]	28nm CMOS	Pyramidal LK	1920 × 1080	125000	30 fps	213 mW	56.8 nJ/pixel
Sinha et al. [21]	GPU	Pyramidal LK	1024 × 768	1000	30 fps	108 W(*)	3600 μJ/pixel(*)

(*) For implementations where it was not reported, the power consumption is estimated around 80% of the platform's Thermal Dissipation Power.

itself. Of course, for a complete system including memory and camera, the performance may be constrained by these elements rather than by the computing core.

The presented data indicates that our architecture operates at the lowest power and at the lowest energy per tracked pixel, when considering that [20] operates at a high keypoint density and should be compared to our dense flow estimation. Additionally, the flexibility of our design enables a wide application range, with frame rates reaching from 5 to 170 fps for different accuracy and density settings. Furthermore, we argue that the scalability allows embedded applications to adapt at-runtime, so they can always work in an optimal operation point.

To our knowledge, this is the first platform to enable low-power, scalable optical flow. As such, it opens up a new application area of optical flow for embedded devices.

V. Conclusion and Future Work

This work demonstrated the design and implementation of a scalable, yet low-power optical flow estimation platform. We have described a scalable approach to optical flow in terms of denseness and accuracy and we elaborated on the custom processor that has been developed based on four innovations.

We demonstrated that the designed processor is fully flexible and can compute optical flow for a variety of Lucas-Kanade implementations. As such, our system enables optical flow scalability, trading flow density and/or accuracy for throughput. For visual navigation, for example, optical flow can be performed at low density and accuracy until a dangerous or interesting pattern is detected. Upon such event, dense optical flow fields can be derived in certain image sections, for very precise navigation (e.g. in case of collision danger). Our flexible platform does allow this high degree of scalability to achieve high throughput, high density and low power consumption all in the same platform.

To our knowledge, this is the first flexible optical flow implementation that is aimed towards low power consumption and it achieves a power efficiency of 15.6 nJ/pixel at highest density and accuracy. This result is two orders of magnitude more efficiency than traditional approaches on CPU or GPU.

Future work will explore applications that can exploit the dynamic optical flow scalability to work at a minimal energy consumption at all times. We are currently exploring methods to visually navigate micro UAVs, which have very strict power budgets, based on the introduced scalable optical flow system.

Acknowledgment

The authors want to acknowledge Synopsys for their support with the ASIP Designer tool.

References

[1] M. Krishnan, M. Wu, Y. Kang, and S. Lee, "Autonomous Mapping and Navigation Through Utilization of Edge-Based Optical Flow and Time-to-Collision," in Innovations and Advances in Computing, Informatics, Systems Sciences, Networking and Engineering, ser. Lecture Notes in Electrical Engineering, T. Sobh and K. Elleithy, Eds. Springer International Publishing, 2015, vol. 313, pp. 149–157.

[2] A. Giachetti, M. Campani, and V. Torre, "The use of optical flow for road navigation," Robotics and Automation, IEEE Transactions on, vol. 14, no. 1, 1998, pp. 34–48.

[3] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite," in Conference on Computer Vision and Pattern Recognition (CVPR), 2012.

[4] J. Marzat, Y. Dumortier, and A. Ducrot, "Real-Time Dense and Accurate Parallel Optical Flow using CUDA," in WSCG 2009: Full Papers Proceedings: The 17th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision in co-operation with EUROGRAPHICS: University of West Bohemia. Václav Skala - UNION Agency, 2009, pp. 105–112.

[5] F. Barranco, M. Tomasi, J. Diaz, M. Vanegas, and E. Ros, "Parallel Architecture for Hierarchical Optical Flow Estimation Based on FPGA," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 20, no. 6, June 2012, pp. 1058–1067.

[6] N. Roudel, F. Berry, J. Serot, and L. Eck, "Hardware Implementation of a Real Time Lucas and Kanade Optical Flow." DASIP, September 2009.

[7] V. Mahalingam, K. Bhattacharya, N. Ranganathan, H. Chakravarthula, R. Murphy, and K. Pratt, "A VLSI Architecture and Algorithm for Lucas Kanade-Based Optical Flow Computation," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 18, no. 1, Jan 2010, pp. 29–38.

[8] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: an efficient alternative to SIFT or SURF," in Computer Vision (ICCV), 2011 IEEE International Conference on. IEEE, 2011, pp. 2564–2571.

[9] B. K. Horn and B. G. Schunck, "Determining Optical Flow," vol. 0281, 1981, pp. 319–331.

[10] J. Revaud, P. Weinzaepfel, Z. Harchaoui, and C. Schmid, "EpicFlow: Edge-preserving interpolation of correspondences for optical flow," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1164–1172.

[11] B. D. Lucas and T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision," in Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2, ser. IJCAI'81. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1981, pp. 674–679.

[12] J. Bouguet, "Pyramidal implementation of the Lucas Kanade feature tracker," Intel Corporation, Microprocessor Research Labs, 2000.

[13] T. Senst, V. Eiselein, and T. Sikora, "II-LK A Real-Time Implementation for Sparse Optical Flow," in Image Analysis and Recognition, ser. Lecture Notes in Computer Science, A. Campilho and M. Kamel, Eds. Springer Berlin Heidelberg, 2010, vol. 6111, pp. 240–249.

[14] A. Plyer, G. Le Besnerais, and F. Champagnat, "Massively parallel Lucas Kanade optical flow for real-time video processing applications," Journal of Real-Time Image Processing, 2014, pp. 1–18.

[15] S. Baker, D. Scharstein, J. P. Lewis, S. Roth, M. J. Black, and R. Szeliski, "A Database and Evaluation Methodology for Optical Flow," International Journal of Computer Vision, vol. 92, no. 1, 2010, pp. 1–31.

[16] S. Smets, T. Goedemé, and M. Verhelst. Scalable Lucas-Kanade optical flow. [Online]. Available: http://nl.mathworks.com/matlabcentral/fileexchange/58843-scalable-lucas-kanade-optical-flow (2016)

[17] J. L. Barron, D. J. Fleet, and S. S. Beauchemin, "Performance of optical flow techniques," International Journal of Computer Vision, vol. 12, no. 1, pp. 43–77.

[18] Y. Murachi, Y. Fukuyama, R. Yamamoto, J. Miyakoshi, H. Kawaguchi, H. Ishihara, M. Miyama, Y. Matsuda, and M. Yoshimoto, "A VGA 30-fps Realtime Optical-Flow Processor Core for Moving Picture Recognition," IEICE Transactions on Electronics, vol. E91.C, no. 4, 2008, pp. 457–464.

[19] M. Komorkiewicz, T. Kryjak, and M. Gorgon, "Efficient hardware implementation of the Horn-Schunck algorithm for high-resolution real-time dense optical flow sensor," Sensors, vol. 14, no. 2, 2014, pp. 2860–2891.

[20] Y. Sasagawa and A. Mori, "High-level video analytics pc subsystem using soc with heterogeneous multi-core architecture," in 2015 Symposium on VLSI Circuits (VLSI Circuits), June 2015, pp. C148–C149.

[21] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics hardware," Machine Vision and Applications, vol. 22, no. 1, 2011, pp. 207–217.